TRS-80 BASIC PROGRAM UTILITY
GENERALIZED SUBROUTINE FACILITY
"GSF"
USERS MANUAL

Written For RACET computes By

T.S. JOHNSTON
and
R.D. JOHNSTON

For Use On The Radio Shack® TRS-80™
Level II BASIC 16-48K Microcomputer System

IMPORTANT NOTICE

ALL RACET computes programs are distributed on an "AS IS" basis without warranty. Neither RACET computes nor the contributor makes any express or implied warranty of any kind with regard to this program material, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. Neither RACET computes nor the contributor shall be liable for incidental or consequential damages in connection with or arising out of the furnishing, use or performance of this program material.

TRS-80 BASIC PROGRAM UTILITY
GENERALIZED SUBROUTINE FACILITY
"GSF"

INTRODUCTION

GSF is a system for incorporating machine language programs in a unified structure that provides easy access by TRS-80 BASIC users. Included with GSF is a series of very useful utility subroutines. These subroutines extend the power of the TRS-80 by rapidly performing functions that would be slow or impractical to do directly in BASIC. GSF resides in upper protected memory.

GSF may be easily expanded by the user to include additional functions. Instructions are provided documenting the conventions used. RACET computes has additional products incorporating or utilizing GSF.

The following summarizes the utility subroutines provided with GSF. A detailed list of subroutines is contained in Appendix(A).

- Display Screen Control
  Five subroutines are provided for scrolling the screen up, down, left, right, and for inverse graphic video. This can add impact to screen displays.

- Draw Horizontal and Vertical Lines
  These two subroutines draw horizontal and vertical graphic lines of any length and location on the screen. The use of these subroutines dramatically decrease display times.

- Duplicate Memory
  Two subroutines are provided to duplicate a byte in memory. This is useful for setting arrays to zero or rapidly placing rows or columns of repeated characters on the screen.

- Move Data
  This subroutine moves data from one location in memory to another. This can be used to rapidly set one array of data equal to another or to move data into protected memory. The latter option provides a "common" area that can be passed from one BASIC program to another.

- Compress and Uncompress Data
  Two subroutines are provided which compress data in memory by removing repeated characters with the ability to uncompress the data to the original form. This is useful in saving and subsequent regeneration of screen images or other data in the minimum space possible. This, coupled with the Read and Write Tape Data Subroutines, provides an efficient method of data storage and retrieval.

- Read and Write Tape Data
  Two subroutines are provided to read and write data to cassette tape. This provides the user with the capability of reading an entire array or screen image with one command. No leaders are written between data items thus significantly reducing read/write times. Data validity checking is performed to ensure correct data is read, thus eliminating many data input/output errors.

- In-Core Sort
  Two subroutines are provided for sorting data in memory. The first subroutine sorts records consisting of corresponding elements of up to 15 arrays using multiple ascending or descending sort keys. The second subroutine sorts records consisting of elements of a character string array using multiple substrings as ascending and/or descending sort keys. This sort system is very fast, versatile, and easy to use.

METHOD OF OPERATION

GSF is a SYSTEM program residing in upper protected memory along with GSF subroutines. GSF provides the interface between the BASIC user and the machine language GSF subroutines. GSF allows BASIC to:

- Determine the number of arguments to be passed to a GSF subroutine.
- Receive and save arguments required.
- Pass control to the specified GSF subroutine

In addition to the above, a common work area is provided for all GSF subroutines, thus minimizing storage requirements. Control is passed to GSF by a series of USR calls as described in the following sections.

LOADING GSF

GSF is loaded into memory using the TRS-80 SYSTEM command. The following steps should be followed to load GSF from tape. Note that all user input is shown underlined.

1. Power up the system. The power switch can be used or can be simulated on systems without the expansion interface by entering the following:

   SYSTEM
   ?/0

2. Set the memory size when requested:

   MEMORY SIZE? 29950 (For a 16K system - use 45784 for 32K systems, 62168 for 48K systems)

3. Prepare the cassette recorder with the GSF tape. See Appendix(B) for a discussion on tape usage.

4. Execute the following commands:

SYSTEM	(User enters SYSTEM)
?GSF	(User enters GSF when requested)
?/	(After tape loads user enters a /)
READY	(The system responds with READY)
CLEAR	(User then enters CLEAR)

5. GSF is now ready for use as described in the subroutine documentation that follows.

GSF can be used with DOS systems by loading from tape as described above, or by loading to disk and then to memory. The procedures for loading to disk and subsequently to memory are described in detail in Appendix(C).

SUBROUTINE EXECUTION

Each GSF subroutine is executed by performing a series of "USR" function calls. The USR calls pass the subroutine number to be executed, integer data, or VARPTR information to GSF. A specific number of USR calls are required for each GSF subroutine.

The TRS-80 USR function is limited to passing a single argument consisting of a two byte integer(-32768 to +32767). As a result, several GSF subroutines require that an address of a data item be passed as an argument(which is also a two byte integer). The VARPTR function is used for this purpose as described in the next section. The mode of the argument required will be clearly specified in the documentation for each GSF subroutine.

The first USR function always specifies the subroutine number to be executed. Any additional USR function calls pass the data or addresses required for each argument. A single BASIC statement for each USR function call could be issued. However, it is HIGHLY RECOMMENDED that ALL USR functions required for a given GSF subroutine be placed in ONE BASIC statement as follows:

var=USR(n_1) OR USR(arg#1) OR USR(arg#2) ... OR USR(arg#n)

where:

var	represents an Integer variable used to store a return value(if any required) from the GSF subroutine.
arg#1,2, ...n	represents Integer numbers, variables, absolute addresses, or VARPTR functions for values being passed to GSF.

The use of the above technique for calling GSF subroutines is especially important when the VARPTR function is used to supply an address argument. Figure(1) illustrates the above recommended procedure.

Purpose: Move the contents(404 bytes) of an array "A" to array "B".

Program:

```
10   DIM A(100),B(100)
20   FOR I=0 TO 100            :REM INITILIZE ARRAY "A"
30     A(I)=I
40   NEXT
:                              :REM OTHER PROGRAM STATEMENTS
:
500  J=USR(14) OR USR(VARPTR(A(0))) OR USR(VARPTR(B(0))) OR USR
(404)
:
:
```

Comments: Lines 10-40 illustrate definitions of arrays "A" and "B", and initilization of array "A". Line 500(a single line) contains the call to GSF subroutine #14 which moves data from one location to another. The three additional arguments specify the source, destination, and quantity of data to be moved. Refer to the documentation for specific details on the arguments and return values for GSF #14.

Figure(1). Example Recommended Calling Procedure.

ARGUMENT USAGE

Given below is a discussion of the special considerations for arguments used with GSF.

A. Integer Arguments: All integers in the TRS-80 system are expresed as two byte(16-bit) signed values in two's complement notation (-32768 to +32767). Integers or variables declared to be integers may be used, for example:

```
10  DEFINT I-N              :REM  DECLARES I THRU N VARIABLES
20  INPUT I,J,K             :REM  AS INTEGERS
30  K=USR(I) OR USR(J) OR USR(K)
```

The DEFINT statement MUST be used for variables used as arguments.

B. Address Arguments: Address data is required by some GSF subroutines. The address can be expressed as follows:

1. Absolute Address - Addresses within the TRS-80 system are expressed as two-byte(16 bit) unsigned integers(0 to 65535). Addresses between 32768 and 65535 must be expressed as negative numbers. The address specifications are summarized below.

Required	Expressed As	Required	Expressed As
0	0	32768	-32768
1	1	32769	-32767
:	:	:	:
:	:	:	:
32767	32767	65535	-1

2. VARPTR Address - The VARPTR function is provided to supply an address of a variable or element of an array, and must be used with special care. The argument of the VARPTR function is the variable name or array name with a subscript, ie:

   VARPTR(WT) or VARPTR(A(0))

   All USR functions for a given GSF subroutine should be placed in one BASIC statement as described in the preceeding section. This is especially important when VARPTR addresses are used.

Special restrictions apply to the use of character strings with GSF and should be used only when specifically directed.

The mode for each argument is indicated for each GSF subroutine. Care must be taken to ensure that the mode and number of arguments are correct for the GSF subroutine being used.

GSF SYSTEM REINITILIZATION

GSF requires that a sequence of USR functions be issued in a specific order. If a bad argument is passed to a USR function and the BASIC program issues an error message GSF is still expecting the GSF sequence to continue. If the user issues a RUN statement GSF is not automatically reinitilized and will most likely cause GSF to be out of synchronization causing unpredictable results.

GSF may be reinitilized by setting the last byte of memory to zero as follows:

```
POKE 32767,0          (for a 16K system)
POKE -16385,0         (for a 32K system)
POKE -1,0             (for a 48K system)
```

It is recommended that the POKE statement be placed at the start of every program using GSF to ensure that GSF is initialized.

GSF SUBROUTINE SPECIFICATION AND EXAMPLES

Each of the seven classes of GSF subroutines available are described below. The purpose, calling sequence, comments, and examples for each class are presented.

A. Display Screen Control

Purpose: The purpose of these five subroutines is to shift(scroll) the contents of the display screen one column left or right, one row up or down, or to invert the video display for any graphic or blank character. These functions can be used for special effects, such as a moving bar chart that moves from right to left.

Calling Sequence:

GSF#	Arg#	Mode	Description
5			Scroll Screen Up
	None	-	No arguments required.
		Return Value	0
6			Scroll Screen Down
	None	-	No arguments required.
		Return Value	0
7			Scroll Screen Left
	None	-	No arguments required.
		Return Value	0
8			Scroll Screen Right
	None	-	No arguments required.
		Return Value	0
1			Reverse Graphic Video
	1	Address	Location of the graphic data to be inverted (Usually between 15360 and 16383).
	2	Integer	Number of bytes to be inverted.
		Return Value	Address+1 of the last byte inverted.

Comments: The first four subroutines require no arguments. A blank row or column is inserted at the opposite end of the shift. Combinations provide for shifting diagonally, ie to the upper right:

```
10 I=USR(5) OR USR(8)
```

The reverse video will only invert graphic or blank characters. Each graphic location will be changed from white to black or reverse. Blanks are treated as graphic black characters and are also inverted. Applying the reverse video command twice to the same location converts the display to its original form.

Example #1: This example illustrates the four scrolling subroutines (GSF #5-8). A message is printed in the lower left corner of the screen, shifted clockwise several times, and then diagonally to the upper right off the screen. Line #134 illustrates that it takes four shifts right or left to equal one shift up or down.

```
100  REM                              :SET DISPLAY LOWER LEFT CORNER
102  CLS
104  PRINT @640,"************"
106  PRINT @704,"*          *"
108  PRINT @768,"* SHIFT IT *"
110  PRINT @832,"* CLOCKWISE*"
112  PRINT @896,"*          *"
114  PRINT @960,"************"
116  REM                              SHIFT CLOCKWISE 5 TIMES
118  FOR I=1 TO 5
120    FOR J=1 TO 9:   K=USR(5):   NEXT        :REM  UP 9
122    FOR J=1 TO 51:  K=USR(8):   NEXT        :REM  RIGHT 51
124    FOR J=1 TO 9:   K=USR(6):   NEXT        :REM  DOWN 9
126    FOR J=1 TO 51:  K=USR(7):   NEXT        :REM  LEFT 51
128  NEXT
130  REM                              SHIFT DIAGONALLY UP & RIGHT
132  FOR I=1 TO 16
134    K=USR(8) OR USR(8) OR USR(5) OR USR(8) OR USR(8)
136  NEXT
138  GOTO 31
```

Example #3: The purpose of this example is to illustrate the use of the scroll left subroutine(GSF #7) for generating a moving bar chart. Line #226 scrolls the screen left one column, leaving room for the next bar to be plotted. Line #216 draws the vertical bar using GSF #12(See Section(B)). Standard basic statements are used to put in the hash marks and bottom scale.

```
200  CLS: DEFINT I-N
202  IC=1:  NV=21:  NM=NV/2+1:  IN=1:  IS=0
204  REM                              GENERATE BAR HEIGHTS 10 TIMES
206  FOR LL=1 TO 20:  FOR J=1 TO 10
208    IC=IC+RND(NV)-NM               :REM  USING RANDOM DATA
210    IF IC < 0 THEN IC=0                  TO GENERATE BAR HEIGHTS
212    IF IC > 44 THEN IC=44
214    IS=44-IC
216    K=USR(12) OR USR(IS) OR USR(127) OR USR(IC)
218    REM                            PUT IN HORIZONTAL HASH MARKS
220    SET(127,44):  SET(126,44):  SET(127,0):  SET(126,0)
222    SET(126,11):  SET(126,22):  SET(126,33)
224    REM                            MOVE BAR CHART ONE COLUMN LEFT
226    K=USR(7)
228  NEXT
230  REM                              PUT IN 10-UNIT VERT HASH MARK
232  FOR J=0 TO 47 STEP 2:  SET(126,J):  NEXT
234  REM                              PRINT BOTTOM SCALE
236  PRINT @1018,IN;:  IN=IN+1:  NEXT
238  GOTO 31
```

B. Horizontal and Vertical Line Subroutines

Purpose: Two subroutines are provided for drawing horizontal and vertical lines on the screen. Input specifications are similar to the SET command with a row number and column number. An additional specification includes the length of the line to be drawn. The benefits of these subroutines are the ease with which lines may be specified and the speed inherent in a machine language routine.

Calling Sequence:

GSF#	Arg#	Mode	Description
12			Draw Vertical Line
	1	Integer	Row number at which the vertical line is to start. This number ranges from 0 to 47.
	2	Integer	Column number at which the vertical line is to start. This number ranges from 0 to 127.
	3	Integer	Length of the line in horizontal graphic spaces. This number ranges from 1 to 128.
		Return Value	0
13			Draw Horizontal Line
	1	Integer	Row number at which the horizontal line is to start. This number ranges from 0 to 47.
	2	Integer	Column number at which the horizontal line is to start. This number ranges from 0 to 127.
	3	Integer	Length of the line in vertical graphic spaces. This number ranges from 1 to 48.
		Return Value	0

Comments: Care must be taken to ensure that the line does not exceed the boundries of the screen. For example:

```
10 I=USR(13) OR USR(47) OR USR(127) OR USR(2)
```

would exceed the lower right corner of the screen by one space. No checking is performed on the coordinates specified so that the user can destroy valuable system or program areas causing unpredictable results.

Example #3: This example demonstrates the use of the line drawing subroutines by construction of a grid of lines with numbers printed within each box. Line #326 & 332 are used to draw the horizontal and vertical lines spaces six rows and 32 columns on the screen. Line #348 prints numbers in the boxes using standard BASIC.

```
300  DEFINT I-N:  FOR IL = 1 TO 5
302  GOSUB 340                    :REM  GENERATE THE DATA
304  GOSUB 320                    :REM  GENERATE THE GRIO
306  GOSUB 310                    :REM  DELAY FOR VIEWING PURPOSES
308  NEXT;  GOTO 31
310  REM                          :DELAY SUBROUTINE
312  FOR I=1 TO 500:  NEXT:  RETURN
320  REM                          ;DISPLAY BOXES
324  FOR I=0 TO 47 STEP 6
326    K=USR(13) OR USR(I) OR USR(0) OR USR(128)
328  NEXT
330  FOR J=0 TO 127 STEP 32
332    K=USR(12) OR USR(0) OR USR(J) OR USR(48)
334  NEXT
336  RETURN
340  CLS                          :REM  PRINT DATA SUBROUTINE
342  FOR I=6 TO 63 STEP 16
344    FOR J=64 TO 1023 STEP 128
346      K=I+J
348      PRINT @K,K
350    NEXT
352  NEXT
354  RETURN
```

Example #4: This example shows the use of the line drawing subroutines by drawing centered lines and boxes on the screen. Lines #422-428 draw double width centered lines. Lines #436-444 draw a series of boxes within boxes.

```
400  DEFINT I-N;  FOR IL= 1 TO 5
402  GOSUB 420:  GOSUB 310:      :REM  GENERATE 8 BOXES & DELAY
404  NEXT;  GOTO 31
420  FOR N1 = 1 TO 10: CLS:  IT=0 :REM DRAW .CENTERED LINES
422  J=USR(13) OR USR(23) OR USR(0) OR USR(128)
424  J=USR(13) OR USR(24) OR USR(0) OR USR(128)
426  J=USR(12) OR USR(0) OR USR(63) OR USR(48)
428  J=USR(12) OR USR(0) OR USR(64) OR USR(48)
430                               :REM  THEN DRAW 8 BOXES
432  FOR N=1 TO 8
434  IB=47-IT:  LI=IB-IT+1:    JR=127-JL:  LJ=JR-JL+1
438    J=USR(13) OR USR(IT) OR USR(JL) OR USR(LJ)
440    J=USR(12) OR USR(IT) OR USR(JL) OR USR(LI)
442    J=USR(13) OR USR(IB) OR USR(JL) OR USR(LJ)
444    J=USR(12) OR USR(IT) OR USR(JR) OR USR(LI)
446    IT=IT+3:  JL=JL+8
448  NEXT N
450  RETURN
```

C. Duplicate Memory Subroutines

Purpose: Two subroutines are provided for propogating a byte through memory. Starting at a given location in memory the byte at that location is repeated a specified number of times. The first subroutine repeates the byte by incrementing the memory location by one. The second subroutine performs the same operation but incrementing by 64. The later subroutine is useful for drawing repeated characters vertically on the screen. The first subroutine is useful for drawing horizontal characters on the screen or for zeroing arrays quickly.

Calling Sequence:

GSF#	Arg#	Mode	Description
4			Duplicate Memory Serially
	1	Address	Memory location at which the byte to be duplicated is contained. The next N-1 bytes in increments of one will be set equal to this byte.
	2	Integer	Number of bytes in the duplicated area(N).
		Return Value	Address of the last byte duplicated +1.
9			Duplicate Memory Incrementally by 64
	1	Address	Memory location at which the byte to be duplicated is contained. The next N-1 bytes in increments of 64 will be set equal to this byte.
	2	Integer	Number of bytes in the duplicated area(N).
		Return Value	Address of the last byte duplicated +1.

Comments: Care must be taken not to exceed the limits intended when duplicating bytes through memory. No checking is performed to verify the validity of the request.

The later subroutine(GSF#9) is intended specifically for drawing repeated vertical characters on the screen. For this purpose the range of 15360 to 16383 may be used. Since every row is separated by 64 bytes this subroutine will create a vertical column on the screen. The first subroutine can also be used for drawing horizontal rows across the screen.

The first subroutine is especially useful for clearing an array to zero. However, no other value may be duplicated in this manner. In this case the first element of the array to be cleared must be set to zero prior to using GSF#4. The POKE command may be used for this initilization as shown in Line #526.

Example #5: This example shows the use of GSF#4 & 9 for drawing rows and columns of characters on the screen, and GSF#1 for inverting the video display. In this case a decrementing digit is placed on the screen, propogated horizontally by GSF#4, and vertically by GSF#9. Note that it was possible to use only one call to GSF#4 (Line 506) to create 8 rows on the top half of the screen. It takes several calls to GSF#9 to create the vertical columns(Line #512). Note the use of the POKE command at Lines #504 & 510 for initilization of the first byte prior to the GSF#5 or 9 commands.

```
500  CLS
502  FOR I=57 TO 48 STEP -1      :REM  GENERATE DECREMENTING DIGIT
504    POKE 15360,I              :REM  STORE FIRST DIGIT FOR ROW
506    K=USR(4) OR USR(15360) OR USR(512)
508    FOR K=15880 TO 15935 STEP 16
510      POKE K,I                :REM  STORE FIRST DIGIT FOR COLUMN
512      L=USR(9) OR USR(K) OR USR(8)
514    NEXT
516    GOSUB 310                 :REM  DELAY FOR VIEWING PURPOSES
518  NEXT
520                              :REM  NOW FLASH THE SCREEN
522  FOR M=1 TO 5
524    CLS:  FOR IQ=1 TO 100:  NEXT
526    POKE 15360,191
528    J=USR(4) OR USR(15360) OR USR(1024)
530    FOR IQ=1 TO 100:  NEXT
532  NEXT
534  GOTO 31
```

Example #6: This example illustrates the speed at which an array can be set to zero relative to the normal BASIC technique. A two byte integer array, a single precicion, and double precision array are set to zero first using BASIC then GSF#4. Note that all arrays have one more element than the dimension size due to the presence of element "0". Each Integer element is two bytes, single precision four bytes, and double precision eight bytes long. The lengths required for the arrays each dimensioned to "250" are 502, 1002, and 2004 respectively. Lines #614-618 illustrate the GSF#4 calls required. The VARPTR function is used to specify the location of each array.

```
600  CLS: DEFINT I-N:  DEFSNG S:  DEFDBL O
602  DIM IA(250),SA(250),OA(250)
604  PRINT "SET ARRAYS TO ZERO USING BASIC"
606  FOR I=1 TO 250
608    IA(0)=0:  SA(I)=0:  OA(I)=0
610  NEXT
612  PRINT "DONE - NOW SET TO ZERO USING GSF"
614  I=USR(4) OR USR(VARPTR(IA(0))) OR USR(502)
616  I=USR(4) OR USR(VARPTR(SA(0))) OR USR(1004)
618  I=USR(4) OR USR(VARPTR(DA(0))) OR USR(2008)
620  PRINT "DONE": FOR I = 1 TO 1000:  NEXT
622  GOTO 31
```

D. Move Data Subroutine

Purpose: The purpose of this subroutine is to move data from one location to another. This is useful for setting one array equal to another, moving data or arrays into protected memory(a common area) and back to user memory, and for setting all elements of an array to the same value.

Calling Sequence:

GSF#	Arg#	Mode	Description
14			Move Data
	1	Address	Address from which data is to be moved.
	2	Address	Address to which data is to be moved.
	3	Integer	Number of bytes of data to be moved.
		Return Value	Address+1 of the last byte of data moved (ie, Arg#1 + Arg#3).

Comments: It is the users responsibility to ensure that the addresses specified and the number of bytes to be moved are valid. Unpredictable results will occur if array pointers, system constants, or other vital areas are destroyed.

The user should also be cautioned on the use of the VARPTR function as elaborated in the Arguments Usage Section. The VARPTR address returned is valid only as long as no additional variables are created and stored in the symbol/value table.

A very important use of this subroutine is to save data in a common area to be passed to independent BASIC programs. To use this feature the user must set the MEMORY SIZE option upon initilization of BASIC. The value set not only depends on the size of the GSF system, but also on the amount of common area needed by the user. In this case the common area would be specified by use of an absolute address as explained in the Arguments Usage Section.

This subroutine can also be used to set one array equal to another. In this case VARPTR addresses specify the element in each array at which to start. The number of bytes to be moved must be calculated from the number of elements in the array and the size of each element(Integer=2 bytes, Single Precision=4 bytes, and Double Precision=8 bytes for each element). Character strings must NOT be moved using this technique.

A given element in an array may be propogated throughout the array by use of this subroutine. This is a very quick way of setting all elements of an array equal to a constant value. The address specified in Arg#2 must be 2,4, or 8 larger than the address in Arg#1(depending on the type of array). The number of bytes to move will be 2,4, or 8 smaller than the number of bytes in the array.

Special care must be taken to ensure that the move operations do not exceed the boundries of the arrays specified. No checking for limits is performed by GSF. If the limits are exceeded important system information may be destroyed causing unpredictable results.

Example #7: This example illustrates the use of the Move Data Subroutine, GSF #14. This includes:

- Setting an array to a constant. This is shown in Lines 706-710. Note that the first element of the array was set to the constant desired (1.23). This was then propogated throughout the array. In this case, 1000 bytes were moved (251*4-4).
- Setting one array equal to another. This is shown in Lines #712-714. In this case 1004 bytes were moved starting at element "0" through "250" (251*4).
- Moving an array to a common area. This is shown in Lines #716-718. The common area is normally protected by MEMORY SIZE, e.g., MEMORY SIZE set to 25616 (16K; absolute address 25616), 42000 (32K; absolute address -23536), or 58384 (48K; absolute address -7152).
- Moving an array from a common area back to user memory. This is shown in Lines #726-728. The data saved in the preceeding step are moved to a new array "SC".

```
700  CLS:  CLEAR
702  PRINT "CLEAR MEMORY & CREATE ARRAYS SA & SB"
704  OIM SA(250), SB(250)
706  PRINT "SET SA TO 1.23"
708  SA(0)+1.23
710  I=USR(14) OR USR(VARPTR(SA(0))) OR USR(VARPTR(SA(1))) OR USR
     (1004)
712  PRINT "SET SB=SA"
714  I=USR(14) OR USR(VARPTR(SA(0))) OR USR(VARPTR(SB(0))) OR USR
     (1004)
716  PRINT "MOVE SB TO COMMON AREA"
718  I=USR(14) OR USR(VARPTR(SB(0))) OR USR(25616) OR USR(1004)
720  PRINT "CLEAR MEMORY & CREATE ARRAY SC"
722  CLEAR
724  DIM SC(250)
726  PRINT "MOVE COMMON DATA TO SC"
728  I=USR(14) OR USR(25616) OR USR(VARPTR(SC(0))) OR USR(1002)
730  PRINT"DONE":FOR I=1 TO 2000:  NEXT:   GOTO 31
```

Example #8: This example illustrates saving screen displays in a common area. Lines #804 & 808 save two displays in common area. Lines #814 & 818 alternately display the two screen formats. Note that the display is generated almost instantaneously.

```
800  CLS: GOSUB 320:              :REM  GENERATE BOXES
802  REM                          :SAVE IN PROTECTED MEMORY
804  I=USR(14) OR USR(15360) OR USR (25616) OR USR(1024)
806  CLS:  GOSUB 340              :REM  GENERATE OATA
808  I=USR(14) OR USR(15360) OR USR (26640) OR USR (1024)
810  REM                          :ALTERNATE OISPLAYS WITH DELAY
812  FOR IL=1 TC 5:  GOSUB 310
814  I=USR(14) OR USR(25616) OR USR(15360) OR USR(1024)
816  GOSUB 310                    :REM  OELAY
818  I=USR(14) OR USR(26640) OR USR(15360) or USR(1024)
820  NEXT:  GOTO 31
```

E. Compress and Uncompress Data Subroutines

Purpose: The purpose of these subroutines is to create a compressed copy of an area in memory by eliminating repeated characters. In particular, the screen display often contains many blanks or repeated characters that could be compressed into a smaller amount of memory and then saved along with other screen displays in memory, tape, or disk. The uncompress subroutine reverses the process.

Calling Sequence:

GSF#	Arg#	Mode	Description
10			Compress Data
	1	Address	Address of the data to be compressed.
	2	Address	Address where the compressed data is to be placed. This area will, in general, be smaller than the source area. However, in extreme cases with no repeated characters this area could be larger by one byte for every 256 characters being compressed.
	3	Integer	Length in bytes of the area to be compressed.
		Return Value	Length in bytes of the compressed data created by this subroutine.
11			Uncompress Data
	1	Address	Address of the compressed data to be uncompressed.
	2	Address	Address where the uncompressed data is to be placed.
		Return Value	Length in bytes of the uncompressed data created by this subroutine.

Comments: This subroutine eliminates only repeated characters that are adjacent to each other, such as contiguous blanks horizontally on the display screen. Furthermore, the data to be compressed must be located serially in memory. The primary anticipated use of these subroutines is to manipulate screen displays.

The compressed data may be placed in an array or into protected memory(common area) by the use of this subroutine. Several screen displays could be stored consecutively and recovered almost instantaneously. The value returned by these functions is the length of the compressed or uncompressed data produced. This value can be used to calculate the location where the next data can be placed.

Normal screen displays have a storage requirement 2-3 times smaller when compressed. The area, however, can be larger if no repeated characters are encountered. In this case, an overhead of one byte for every 256 characters being compressed must be considered when allocating storage areas.

Example #9: This example is similar to Example #8 in that two screen displays are generated and saved in a common area. The compress and uncompress subroutines are used, however, to save the screen formats in the minimum amount of space possible. Lines #904 & 912 save the screens in protected memory. The value "I1" returned in Line #904 is the storage length required for the first display. This is used in Line #912 to calculate the starting location for the second display. Lines #918 & 924 alternate the redisplay of these formats. Note that "I2" is used directly in Line #924 although it too could have been calculated from the return value in Line #918.

```
900   CLS:  GOSUB 320                :REM  GENERATE BOXES
902   REM                            :COMPRESS & SAVE IN COMMON
904   I1=USR(10) OR USR(15360 OR USR(25616) OR USR(1024)
906   CLS:  GOSUB 340                :REM  GENERATE OATA
908   REM                            :COMPRESS & ADD TO COMMON
910   I2=I1 + 25616                  :REM CALCULATE NEXT COMMON
912   I3=USR(10) OR USR(15360) OR USR(I2) OR USR(1024)
914   REM                            :NOW ALTERNATE DISPLAYS
916   FOR IL=1 TO 3:  GOSUB 310
918   I=USR(11) OR USR(25616) OR USR(15360)
920   GOSUB 310
922   I=USR(11) OR USR(I2) OR USR(15360)
924   GOSUB 310
926   CLS                            :REM  PRINT STORAGE REQUIREMENTS
928   PRINT "UNCOMPRESSED FORMATS REQUIREO 1024 BYTES"
930   PRINT "FIRST COMPRESSED FORMAT REQUIRED ";I1," BYTES"
932   PRINT "SECOND COMPRESSED FORMAT REQUIRED ",I3," BYTES"
934   GOSUB 310: GOSUB 310: NEXT: GOTO 31
```

Example #10: This example illustrates saving partial screen displays in arrays. Lines #1006 & 1008 save each half of one display and Lines #1014 & 1016 save each half of a second display in a doubly dimensioned array. Note that the order of the dimensions is critical since BASIC stores data by rows (first subscript varies fastest). Line #1022 randomly displays each half display.

```
1000  DIM SO(100,3):  OEFINT I-N :REM  CLEAR & DEFINE MATRIX
1002  GOSUB 340:  GOSUB 320      :GENERATE BOXES & DATA
1004  REM                        :SAVE EACH HALF DISPLAY
1006  I1=USR(10) OR USR(15360) OR USR(VARPTR(SD(0,0))) OR USR(512)
1008  I2=USR(10) OR USR(15872) OR USR(VARPTR(SD(0,1))) OR USR(512)
1010  CLS:  GOSUB 420            :REM  GENERATE DISPLAY #2
1012  REM                        :SAVE EACH HALF DISPLAY
1014  I1=USR(10) OR USR(15360) OR USR(VARPTR(SD(0,2))) OR USR(512)
1016  I2=USR(10) OR USR(15872) OR USR(VARPTR(SD(0,3))) OR USR(512)
1018  REM                        :RANDOMLY DISPLAY EACH HALF
1020  I=RND(4)-1:  J=RND(2)*512+14848
1022  I1=USR(11) OR USR(VARPTR(SA(0,I))) OR USR(J)
1024  GOSUB 310
1026  GOTO 31
```

F. Read and Write Tape Data Subroutines

Purpose: The purpose of these two subroutines is to read and write blocks of cassette tape data. The data written on tape contain no intermediate leaders, thus shortening input/output times. Data validity checking is also performed while reading ensuring data integrity. An ID number is also written with each tape block which can be subsequently checked during read operations.

Data to be written may be of any length but must be located serially in memory. Any portion of the data block may then be read provided sufficient buffer space is made available.

Calling Sequence:

GSF#	Arg#	Mode	Description
2			Read Tape Data Block
	1	Address	Location in memory where data read is to be placed. The area specified must be large enough to hold the maximum tape block expected.
	2	Integer	Maximum amount of data(in bytes) to be read. Tape blocks equal to or shorter than this value will be read successfully. Tape blocks longer than this maximum will be truncated to this value.
	3	Integer	Tape block ID number. This number is compared with the ID number written on the tape. The operation is terminated and the block skipped if a mismatch occurs. Specifying zero for this argument inhibits ID checking(allows any block to be read).
		Return Value	-1 Tape block ID does not match. -2 Data read error occurred while reading the tape block. >0 Number of bytes successfully read.
3			Write Tape Data Block
	1	Address	Location of data in memory to be written to tape.
	2	Integer	Number of bytes to be written.
	3	Integer	Tape block ID. Usage is described in GSF#2, argument #3.
		Return Value	Address+1 of the last byte written(Arg#1 + Arg#2).

Comments: Data to be read or written may be located anywhere within memory. An entire array or screen display may be read or written with one command. This subroutine, coupled with the compress/uncompress subroutines(#10 & 11) provides for efficient data storage and retreival.

Example #11: This example illustrates writing two arrays to tape and reading of the data recorded. Line #1108 creates tape block #1 from a Single Precision array "SA". Line #111B creates tape block #2 from array "SA". Line "1128 is used to read back both blocks. Note that the number of bytes to be written must be calculated from the length of the array and the type (101*4 in this case).

```
1100  CLS:  DIM SA(100)            :REM  CREATE & WRITE #1
1102  FOR I=0 TO 100:  NEXT
1104  PRINT "READY TAPE FOR WRITING BLOCK #1 - PRESS 'Y'"
11D6  T$ = INKEY$:  IF T$<>"Y" THEN 1106
110B  I=USR(3) OR USR(VARPTR(SA(0))) OR USR(404) OR USR(1)
1110  REM                          :CREATE & WRITE #2
1112  FOR I=0 TO 100:  SA(I)=100-I:  NEXT
1114  PRINT "READY TAPE FOR WRITING BLOCK #2 - PRESS 'Y'"
1116  T$=INKEY$:  IF T$<>"Y" THEN 1116
1118  I=USR(3) OR USR(VARPTR(SA(0))) OR USR(404) OR USR(2)
1120  PRINT "REWIND TAPE - SET TO PLAY"
1122  FOR J=1 TO 2
1124  PRINT "PRESS 'Y' TO READ BLOCK #;J
1126  T$=INKEY$:  IF T$<>"Y" THEN 1126
1128  I=USR(2) OR USR(VARPTR(SA(0))) OR USR(404) OR USR(J)
1130  IF I < 0 THEN 1138
1132  FOR I=0 TO 100:  PRINT SA(I);:  NEXT:  PRINT
1134  NEXT
1136  GOSUB 310:  GOTO 31
1138  PRINT "TAPE READ ERROR #;I:  STOP
```

Example #12: This example illustrates saving screen displays on tape in a compressed format. Two screen formats are compressed and written to tape by the subroutine located at Lines #1230-1236. The two tape blocks are read, uncompressed, and displayed on the screen by Lines #121B-1226. If an input/output error occurs during the operation (detected by a negative return code) then Line #1238 is executed.

```
1200  CLS:  DEFINT I-N:  SIM SA(500)
1202  PRINT "TWO SCREEN DISPLAYS ARE GENERATED, COMPRESSED, WRITTEN
TO TAPE, READ, AND ALTERNATELY DISPLAYED"
1204  PRINT "READY TAPE FOR WRITING - 'Y'"
1206  IF INKEY$<>"Y" THEN 1206    :REM  GEN., COMP., & WRITE #1
1208  GOSUB 340:  GOSUB 320:  IB=1:  GOSUB 1230
1210  REM                          :GEN., COMP., & WRITE #2
1212  GOSUB 420:  IB=2:  GOSUB 1230
1214  CLS:  PRINT "REWIND TAPE - SET FOR PLAY - PRESS 'Y'"
1216  T$=INKEY$:   IF T$<>"Y" THEN 1216
1218  FOR J=1 TO 2                 :REM  READ, UNCOMP, & DISPLAY
1220    I=USR(2) OR USR(VARPTR(SA(0))) OR USR(1000) OR USR(J)
1222    IF I   0 THEN 1238
1224    I=USR(11) OR USR(VARPTR(SA(0))) OR USR(15360)
1226  NEXT
1228  FOR I=1 TO 1000: NEXT : GOTO 31
1230  REM                          :COMP. & WRITE ARRAY SUB.
1232  I1=USR(10) OR USR(15360) OR USR(VARPTR(SA(0))) OR USR(1024)
1234  I2=USR(3) OR USR(VARPTR(SA(0))) OR USR(I1) OR USR(IB)
1236  RETURN
1238  PRINT "TAPE READ ERROR #";I:  STOP
```

G. In-Core Sort Subroutines

Purpose: Two subroutines are provided for sorting data in memory. The first subroutine sorts records consisting of corresponding elements of up to 15 arrays using multiple ascending or descending sort keys. The second subroutine sorts records consisting of elements of a character string array using multiple substrings as ascending and/or descending sort keys.

Calling Sequence:

GSF#	Arg#	Mode	Description
17			In-Core Sort - Multiple Variable Mode
	1	Address	Address of the string pointer for sort variable list. This string contains the names of the arrays to be included in the sort, and ascending or descending sort sequence requirements.
	2	Integer	Lower limit index of elements in the array to be sorted.
	3	Integer	Upper limit index of elements in the array to be sorted. The dimension of the array must be at least two greater than this value.
		Return Value	0 Sort completed successfully. 1 Null string passed for Argument #1. 2 Missing variable(trailing comma) in Arg#1. 3 Array specified not found. 4 Array found not singly dimensioned. 5 Array too small.
18			In-Core Sort - Character Variable Mode
	1	Address	Address of the first element of the character string to sort. This array must be singly dimensioned.
	2	Integer	Lower limit index of elements in the array to be sorted.
	3	Integer	Upper limit index of elements in the array to be sorted. The dimension of the array must be at least two greater than this value.
	4	Address	Address of the first element of an Integer array that defines the relative locations, lengths, and ascending/descending attribute of each sort key.
		Return Value	0 Sort completed successfully. 1 Argument #4 array not Integer. 2 Argument #4 array not singly dimensioned. 3 No substrings specified in Argument #4. 4 Substring location zero specified(the first character of a string is specified as "1" - zero does not exist).

Comments: More detailed information on the use of the two sort subroutines is given below. Remember - arguments must be integer.

A. GSF#17 - In this mode of sorting several singly dimensioned arrays of any type are connected element by element and sorted. For example consider the data contained in the following arrays:

	OIM	NM$(7)	SX$(7)	IG(7)	WT(7)
	0	"RON"	"M"	46	165
	1	"ARANA"	"F"	39	103
Section to be sorted	2	"CHRIS"	"M"	15	140
	3	"ERIC"	"M"	18	140
	4	"TAMMY"	"F"	12	95
	5	"SCOTT"	"M"	39	160
	6				
	7				

The arrays to be sorted are specified by Argument#1. This is a character string that contains the names of the arrays in the order of importance for sorting. Array names to be checked for ascending sort sequence are preceeded by a "+", and for descending sort sequence by a "-". Assume that the above data is to be sorted with SX$ as the primary ascending sort key("F" records first,then "M" records), IG descending with NM$ and WT carried along with no checking. Argument #1 could be specified by:

VARPTR(SP$) where SP$="+SX$,-IG,NM$,WT" (Use Ouotes!)

The above example indicates that elements 2-4 only are to be sorted. These limits are specified in Arguments #2 & 3. Note that two unused elements were provided at the end of the arrays. These are REQUIREO for use by the sort subroutine. Elements 0, 1, and 5 not included in the sort will not be disturbed. The complete GSF calling sequence required for the above example is:

I=USR(17) OR USR(VARPTR(SP$)) OR USR(2) OR USR(4)

The return value "I" should be zero indicating a successful sort.

B. GSF#18 - In this mode of sorting a single character string array is sorted. Argument #1 points to the first element of this singly dimensioned array. Arguments #2 & 3 specify the limits of the sort similar to GSF#17. Two extra elements must also be available at the end of the array for use by the sort subroutine.

Argument #4 specifies the location, length, and ascending/descending attributes of each sort key. This information is supplied in an Integer array as shown below:

	0	# keys
Sort Key #1	1	loc
	2	len
Sort Key #2	3	loc
	4	len
	:	:
Sort Key #N	2*N-1	loc
	2*N	len

loc = location of sort key relative to 1. This value is positive for ascending sequence and negative for descending sequence.

len = length of the sort key in bytes.

Example #13: This example illustrates the Multiple Variable mode of sorting using GSF#17. Four arrays, NM$,SX$,IG, and WT, are first initilized with random data. The user is then prompted for the sort parameter specification string. The array is then sorted and printed. Several sorts on the same data can be performed by changing the sort parameter specification string. Note that the arrays are dimensioned to 102 although the data occupies only the elements 0-100, leaving two extra locations needed by the sort subroutine.

```
1300 CLS
1302 CLEAR(3000):  DEFINT I-N    :REM  DEFINE ARRAY & STRING SPACE
1304 DIM NM$(102),SX$(102),IG(102),WT(102)
1306 PRINT "GENERATING ARRAYS"
1308 FOR I=0 TO 100                :REM  ALLOCATE ARRAY & STRING
1310   J=RND(10)                   :REM  RANDOM DATA
1312   FOR K=1 TO J
1314     NM$(I)=NM$(I)+CHR$(RND(26)+64)
1316   NEXT
1318   IF RND(2)=1 THEN SX$(I)="M" ELSE SX$(I)="F"
1320   IG(I)=RND(100)
1322   WT(I)=RND(100)+100
1324 NEXT:  SV$="":  PRINT "INPUT SORT PARAMETERS NM$,SX$,IG,AND WT"
1326 INPUT SV$:  IF LEN(SV$)=0 THEN 31
1328 I=USR(17) OR USR(VARPTR(SV$)) OR USR(0) OR USR(100)
1330 PRINT "SORT COMPLETED, RC=";I
1332 GOTO 1326
```

Example #14: This example illustrates the character variable mode of sorting using GSF#18. An array of 100 elements is first generated. Each element of this array is a character string of 20 bytes. Each 20 byte record is separated [illegible] with a blank at the end of each [illegible] input the location of the [illegible] response might be: 2 Ⓔ-1.1 Ⓒ would sort field defined by [illegible] followed by the field in [illegible]

```
1400 CLS
1402 CLEAR(3000):  DEFINT I-N:  DIM IE(10),S$(100)
1404 PRINT "GENERATING ARRAYS"
1406 FOR I=0 TO 100                REM  GENERATE RANDOM
1408   T$="":  FOR J=1 TO 4
1410     FOR K=1 TO 4:  T$=T$+ CHR$(RND(5)+64):  NEXT:  T$=T$+
1412   NEXT:  PRINT T$:  S$(I)=T$
1414 NEXT
1416 PRINT "INPUT NUMBER OF SORT KEYS ";:  INPUT IE(0)
1418 FOR I=1 TO IE(0)*2 STEP 2
1420   PRINT "INPUT KEY #";(I+1)/2;" LOC & LEN";
1422   INPUT IE(I),IE(I+1)
1424 NEXT
1426 PRINT "SORT STARTING"
1428 J=USR(18) OR USR(VARPTR(S$(0))) OR USR(0) OR USR(100) OR USR
     (VARPTR(IE(0)))
1430 PRINT "SORT COMPLETED, RC=";J
1432 FOR I=0 TO 100:  PRINT S$(I):  NEXT
1434 GOTO 31
```

USER WRITTEN SUBROUTINES

Introduction: The GSF system has been designed to allow the addition of user written subroutines. The purpose of this section is to document the information required for an experienced assembly language programmer to utilize the features of the GSF system. An example is given illustrating the principals involved.

The user must be CAUTIONED that addition of machine language programs can reduce the integrity of the TRS-80 system. The TRS-80 BASIC system has many built-in checks to ensure the valid operation of the system. Machine language programs, unless carefully designed, can cause unpredictable results.

Method of Operation: GSF provides an interface between the user and machine language subroutines utilizing the "USR" function as described in earlier sections of this manual. The general sequence of steps followed by GSF is:

1. GSF receives the first USR function call containing the subroutine number.
2. The subroutine number is fetched and its corresponding three byte transfer vector is located. This vector is of the form:

```
DEFB    n       ;n= # arguments
DEFW    addr    ;addr= address of subroutine to
                ;      be executed.
```

3. If no arguments are required(n=0) control is passed by a jump instruction(JP) to the address specified by "addr". If arguments are required(n>0) then control is passed back to the BASIC user with a return value of zero(this is why the "OR" operator can be used).
4. The user issues as many additional(n) USR function calls as necessary. Each argument is fetched and saved sequentially in an argument storage area by GSF(area GAV in the example).
5. When the last argument is fetched GSF passes control to the routine specified by "addr".
6. The subroutine performs the action required and passes control directly back to the BASIC user with a RET instruction(if no return value is required), or by placing the desired return value in the HL register and returning to BASIC by executing the instruction "JP 0A9AH".

The specific GSF subroutine extracts the needed data directly from the argument storage area. This technique provides a standardized calling sequence and simplifies the efforts required to add machine language subroutines to the TRS-80.

Procedure: The user can add additional subroutines to GSF by following the steps outline below:

1. Write the program section utilizing the argument storage area for arguments or temporary data storage.
2. Add the common description block as shown in the example.
3. Add the transfer vectors required. These should be added from the end backwards(GSF39,38, ... etc.).

4. Assemble the program using the Radio Shack Editor Assembler.

5. Reset protected memory to a lower value than specified for GSF to allow room for the additional subroutines.

6. Load the distributed GSF system FIRST using the SYSTEM command as documented earlier in this manual.

7. Load the new GSF subroutines either immediately after Step 6. above or by use of a separate SYSTEM load. Different sets of new GSF subroutines can be loaded without reloading the initial GSF system.

It is recommended that user written GSF subroutines be assigned numbers 39,38, ... etc. in order to leave room for additional subroutines to be distributed by RACET computes.

GSF occupies the highest memory locations available as shown below:

Memory Locations	Label	Description
29950 - 32464	SMEM	Existing GSF subroutines
32465 - 32640	GAV	Argument storage area
32641 - 32646		Reserved for use by GSF
32647 - 32766	GST	Transfer vector area
32767	EMEM	Last byte of memory.

The above address are for a 16K system. Add 16384 or 32768 for 32K or 48K systems. Additional user subroutines must be placed below SMEM(29950). The area in GAV not used for arguments may be used as a general work area. The GSF system area declaration section given in the example is recommended for including in any user written GSF subroutine.

Example: An example has been included in this section which illustrates the addition of two user written GSF subroutines. These two subroutines have been included for illustration purposes only and are not necessarily recommended for general use. Shown first is the complete assembly language program for adding GSF #38 and 39. This is followed by two basic programs illustrating their purpose and use.

EXAMPLE USER WRITTEN GSF SUBROUTINE

```
EXMEM   EQU     0               ;EXTRA MEMORY AVAILABLE
        ORG     29950+EXMEM-150 ;START 150 BYTES BELOW SMEM
;
;       GSF38   LOCATE ELEMENT IN A SORTED INTEGER ARRAY USING
;               SEQUENTIAL SEARCH
;
;       ARG#1   ADDRESS         ARRAY TO BE SEARCHED
;       ARG#2   INTEGER         VALUE TO BE FOUNO
;       ARG#3   INTEGER         LENGTH OF ARRAY TO BE SEARCHED
;
;       RETURN VALUE            INDEX OF THE FIRST VALUE IN THE
;                               ARRAY WHICH IS EQUAL TO OR JUST
;                               LARGER THAN ARG#2.  IF THE VALUE
;                               IS LARGER THAN THE LAST ELEMENT
;                               IN THE ARRAY THE RETURN VALUE
;                               WILL BE ONE GREATER THAN ARG#3.
;
```

EXAMPLE USER WRITTEN GSF SUBROUTINE(continued)

```
GFS38   LD      IX,(GAV)        ;FETCH ADDRESS OF ARRAY
        LD      BC,(GAV+2)      ;FETCH VALUE TO BE LOCATED
        LD      DE,(GAV+4)      ;FETCH LENGTH OF ARRAY
G38A    LD      H,(IX+1)        ;FETCH ARRAY ELEMENT
        LD      L,(IX)
        OR      A               ;COMPARE WITH ARG#2
        SBC     HL,BC
        JP      PE,G38B
        JP      P,G38D          ;ARG#2 >= ARRAY ELEMENT
        JR      G38C
G38B    JP      M,G38D
G38C    INC     IX              ;ARG#2 < ARRAY ELEMENT
        INC     IX
        DEC     DE
        LD      A,D
        OR      E
        JR      NZ,G38A         ;TRY NEXT ELEMENT
G38D    PUSH    IX              ;CALCULATE INDEX OF ELEMENT
        POP     HL
        LD      DE,(GAV)        ;(FOUND-BEGINNING)/2
        OR      A
        SBC     HL,DE
        SRL     H
        RR      L
        JP      0A9AH           ;RETURN INDEX TO BASIC USER
;
;       GSF39   ENCYPHER/ DECYPHER CHARACTER STRING SUBROUTINE
;
;       ARG#1   ADDRESS         STRING POINTER TO BE ENC/DEC
;       ARG#2   ADDRESS         STRING POINTER TO PASSWORD
;
;       RETURN VALUE            0
;
GSF39   LD      IY,(GAV)        ;ENC/DEC STRING POINTER
        LD      C,(IY)          ;LENGTH OF STRING
        LD      H,(IY+2)        ;ADDRESS OF STRING
        LD      L,(IY+1)
        PUSH    HL              ;SET STRING ADDRESS
        POP     IY
        LD      IX,(GAV+2)      ;PASSWORD STRING POINTER
        LD      E,(IX)          ;FIND PASSWORD LENGTH
        LD      H,(IX+2)        ;FIND PASSWORD ADDRESS
        LD      L,(IX+1)
        LD      (GAV+2),HL      ;REPLACE POINTER WITH ADDRESS
        LD      A,173           ;SET RANDOM SEED IN "A"
G39A    LD      HL,(GAV+2)      ;RESET TO START OF PASSWORD
        LD      B,E             ;AND LENGTH RESET
G39B    ADD     (HL)            ;CREATE RANDOM BYTE SEQUENCE
        RL      A               ;FROM PASSWORD - A BETTER RANDOM
        XOR     B               ;GENERATOR SHOULD BE USED
        INC     HL
        DJNZ    G39B
        LD      D,A             ;ENCYPHER/DECYPHER BYTE
        XOR     (IY)
        LD      (IY),A
        LD      A,D
```

EXAMPLE USER WRITTEN GSF SUBROUTINE(continued)

```
        INC     IY
        DEC     C
        JR      NZ,G39A         ;LOOP FOR ALL BYTES IN STRING
        RET                     ;RETURN TO BASIC USER
;
;       GSF SYSTEM BLOCK DECLARATION SECTION
;
        ORG     29950+EXMEM     ;START OF EXISTING GSF
SMEM    EQU     $               ;SUBROUTINES
        ORG     32465+EXMEM     ;START OF ARGUMENT STORAGE
GAV     DEFS    176             ;AREA AND WORK AREA
        DEFS    6               ;RESERVED FOR GSF
GST     EQU     $               ;START OF TRANSFER VECTORS
        DEFS    114             ;SKIP TO #38(38*3 = 114)
        DEFB    3               ;GSF#38 - THREE ARGUMENTS
        DEFW    GSF38           ;         ENTRY ADDRESS
        DEFB    2               ;GSF#39 - TWO ARGUMENTS
        DEFW    GSF39           ;         ENTRY ADDRESS
EMEM    DEFB    0               ;LAST BYTE OF MEMORY
        END     1A19H           ;AFTER LOAD BRANCH TO BASIC
```

Example Use of GSF#38: The purpose of this subroutine is to search an integer array which is assumed to be sorted in ascending sequence for the first value which is equal to or larger than a specified integer.

```
3800    DEFINT I-N:  POKE 32767,0:  CLS:  DIM IE(10)
3802    J=0:  PRINT @0, "INDEX -";:  PRINT @64,"ARRAY -";
3804    FOR I=0 TO 10
3806      J=J+RND(10):  IE(I)=J
3808      PRINT @I*4+10,I;       :REM  INDEX
3810      PRINT @I*4+74,J;       :REM  ARRAY TO BE SEARCHED
3812    NEXT
3814    PRINT @192,"INPUT VALUE TO BE LOCATED";:  INPUT K
3816    J=USR(38) OR USR(VARPTR(IE(0))) OR USR(K) OR USR(11)
3818    PRINT @320,"INDEX LOCATED=";J
3820    GOTO 3814
```

Example Use of GSF #39: The purpose of this subroutine is to transform a character string into a string that is encyphered (scrambled) by use of a password string. The encyphered string can be decyphered(unscrambled) by the same subroutine and password.

```
3900    CLEAR 500:  POKE 32767,0
3902    PRINT "INPUT PASSWORD";:  INPUT P$
3904    PRINT "INPUT STRING TO BE ENCYPHERED";:  INPUT S$
3906    J=USR(39) OR USR(VARPTR(S$)) OR USR(VARPTR(P$))
3908    PRINT "ENCYPHERED STRING=";S$
3910    J=USR(39) OR USR(VARPTR(S$)) OR USR(VARPTR(P$))
3912    PRINT "DECYPHERED STRING=";S$
3914    GOTO 3902
```

APPENDIX A. — GSF SUBROUTINE SUMMARY

GSF#	Page#	Arg#	Mode	Description
1	5	0		Invert Graphic Video
		1	Address	Graphic data to be inverted
		2	Integer	Number of bytes to be inverted.
			Return Value	0
2	15	0		Read Tape Data Block
		1	Address	Location where data to be placed.
		2	Integer	Maximum number of bytes to be read.
		3	Integer	Tape Block ID number.
			Return Value	-1 Tape block ID does not match. -2 Data read error. >0 Number of bytes read.
3	15	0		Write Tape Data Block
		1	Address	Location of data to be written.
		2	Integer	Number of bytes to be written.
		3	Integer	Tape Block ID number.
			Return Value	Address last byte written +1.
4	9	0		Duplicate Memory Serially
		1	Address	Location of start of data.
		2	Integer	Number of bytes to be duplicated.
			Return Value	Address last byte duplicated.
5	5	0	None	Scroll Screen Up
			Return Value	0
6	5	0	None	Scroll Screen Down
			Return Value	0
7	5	0	None	Scroll Screen Left
			Return Value	0
8	5	0	None	Scroll Screen Right
			Return Value	0
9	9	0		Duplicate Memory Incrementally by 64
		1	Address	Location of start of data.
		2	Integer	Number of bytes to be duplicated.
			Return Value	Address last byte duplicated.
10	13	0		Compress Data
		1	Address	Data to be compressed.
		2	Address	Where compressed data is to be placed.
		3	Integer	Number of bytes to be compressed
			Return Value	Number of bytes in compressed area.

GSF#	Page#	Arg#	Mode	Description
11	13			Uncompress Data
		1	Address	Data to be uncompressed.
		2	Address	Where uncompressed data to be placed.
			Return Value	Number of bytes in uncompressed area.
12	7			Draw Vertical Line
		1	Integer	Row number for vertical line.
		2	Integer	Column number for vertical line.
		3	Integer	Length of vertical line.
			Return Value	0
13	7			Draw Horizontal Line
		1	Integer	Row number for horizontal line.
		2	Integer	Column number for vertical line.
		3	Integer	Length of vertical line.
			Return Value	0
14	11			Move Data
		1	Address	Location of data to be moved.
		2	Address	Location where data is to be moved.
		3	Integer	Number of bytes of data to be moved.
			Return Value	Address last byte +1 (Arg#3+Arg#1).
15				Fetch GSF Argument
		I	Integer	GSF argument # to be fetched.
			Return Value	Integer argument saved by GSF.
16				Fetch Memory Word
		1	Address	Address of memory location fetched.
			Return Value	Integer value at memory location.
17	17			In-Core Sort - Multiple Variable Mode
		1	Address	Pointer to sort key string.
		2	Integer	Start index for sort.
		3	Integer	End index for sort.
			Return Value	0 Sort completed successfully.
				1 Null Argument #1.
				2 Missing variable.
				3 Array specified not found.
				4 Array found not single dimension.
				5 Array too small.
18	17			In-Core Sort - Character String Mode
		1	Address	Pointer to array to be sorted.
		2	Integer	Start index for sort.
		3	Integer	End index for sort.
		4	Address	Sort key parameter list.
			Return Value	0 Sort completed successfully.
				1 Argument #4 array not integer.
				2 Argument #4 array multi-dimension.
				3 No substrings specified.
				4 Substring location 0 specified.

APPENDIX B. -- GENERAL NOTES

TAPE USAGE

Software programs purchased from RACET computes should load at the same volume setting as your standard setting for tapes produced by your computer. It is recommended that you periodically clean the head capstan and pinch rollers on your cassette recorder using commercially available cassette cleaning and demagnetizing accessories. Dirty heads can cause substantial loss of volume and induce unwanted noise.

Machine language tapes loaded using the SYSTEM command are inherently more sensitive to volume settings than BASIC programs loaded by the CLOAD command. The user may need to try several cassette volume levels in order to read the machine language tapes correctly. A "C" in the upper right portion of the video display indicates an incorrect load. The SYSTEM loader checks for data input validity on each block read. This practically ensures that if the tape loads without an indicated error the memory contents will be correct. BASIC tapes, however, perform only minimal error checking. This often results in an apparent successful load (no error messages) but the contents of memory will be bad. The list command can be used after loading a BASIC tape to verify the contents of memory.

RACET computes programs are recorded twice on the same side of the tape. Although this somewhat more expensive than recording both directions on a shorter tape, it was felt that this was to the benefit of the customer. RACET computes tapes are all loaded in a five-screw shell. If a tape breaks for any reason, the cassette shell may be opened, the tape spliced, and the second recording will remain intact with no gaps resulting from splicing.

TAPE CONTENT AND EXAMPLES

General Subroutine Facilities (GSF) object code is located on the first two recordings on the tape. This is the code you will be using with your programs. Following the second recording of GSF are located two BASIC programs incorporating the examples shown in this User Manual (stored as "1" and "2"). The first program includes Examples 1 through 5 and 13 and 14. The second program includes Examples 6 through 12. Following the initial display, each of the example programs provides a menu for selecting the desired displays. The examples in the manual are shown for the 16K version. Differences are only in absolute memory locations as follows:

16K	32K	48K
32767	-16385	-1
25616	-23536	-7152
26650	-22512	-6128

Each example in this manual is shown to branch back to the menu selection portion of the program(GOTO 31).

IF YOU EXPAND YOUR MEMORY

Users increasing the memory size of their systems can order a larger version for just the cost of handling - $5.00. Include your sales receipt, program name, and version required with your request.

APPENDIX C. -- DISK OPERATING SYSTEM PROCEDURES

The following procedures can be used to load and use either the GENERAL SUBROUTINE FACILITIES or DISK SORT PROGRAM machine language code while operating under DOS. All user input is shown underlined with the values 'aaaa', 'bbbb', 'cccc', and 'ddddd' to be inserted as shown in the table below:

SYSTEM	VERSION	aaaa	bbbb	cccc	ddddd
GSF	16K	74FE	7FFF	7E80	29950
	32K	B2D8	BFFF	BE80	45784
	48K	F2D8	FFFF	FE80	62168

A. LOADING AND EXECUTING SYSTEMS FROM TAPE

1. Bring in TRSDOS (user enters BASIC command)
2. BASIC (user enters BASIC command)
3. HOW MANY FILES? (as per user requirements)
4. MEMORY SIZE?ddddd (see 'ddddd' in table above)
5. CMD "T" (turns off clock)
6. SYSTEM (executes SYSTEM command)
7. ?GSF (loads system tape)
8. ?bk (bk = BREAK key)
9. DEFUSR=&Hcccc (see 'cccc' in table above)
10. GSF or DSP object code is now ready for use. Access using USR(arg) commands as described in the user manuals.

B. TRANSFERRING SYSTEMS FROM MEMORY TO DISK

1. Load appropriate system tape as described in above procedure.
2. CMD "S" (returns to TRSDOS mode)
3. DUMP GSF/OBJ (START=X'aaaa',END=X'bbbb',TRA=X'cccc')
4. DIR (to verify GSF/OBJ on disk)

(the name "GSF/OBJ" is not mandatory. "DSP/OBJ" is suggested for the Disk Sort Program)

C. LOADING AND EXECUTING SYSTEMS FROM DISK

1. Bring in TRSDOS (CMD "S" to exit BASIC)
2. LOAD GSF/OBJ (no quotes)
3. BASIC (user enters BASIC command)
4. HOW MANY FILES? (as per user requirements)
5. MEMORY SIZE?ddddd (see 'ddddd' in table above)
6. DEFUSR=&Hcccc (see 'cccc' in table above)
7. CLEAR (must be entered as shown)
8. GSF or DSP machine code is now ready for use. Access using USR(arg) commands as described in the user manuals.

Note that GENERAL SUBROUTINE FACILITIES or DISK SORT PROGRAM object code will remain in protected memory as long as the TRS-80 remains in DOS BASIC. If you transfer from DOS BASIC to the TRSDOS, steps 3-7 in C. above must be repeated when returning to DOS BASIC.

Use normal DOS procedures to load and save programs to disk.